Merge sort

David Kauchak
cs201
Spring 2014

**MergeSort: Merge**

Assuming left (L) and right (R) are sorted already, merge the two to create a single sorted array

L: 1 3 5 8  
R: 2 4 6 7

How can we do this?

**Merge**

L: 1 3 5 8  
R: 2 4 6 7

Create a new array to hold the result that is the combined length

**Merge**

L: 1 3 5 8  
R: 2 4 6 7

What item is first?  
How did you know?
Compare the first two elements in the lists!

What item is second? How did you know?

Compare the smallest element that hasn’t been used yet in each list
- For L, this is next element in the list
- For R, this is still the first element

General algorithm?
**General algorithm:**
- Keep a “pointer” (index) for where we are in each input array
- Start them both at the beginning
- Repeat until we’re done:
  - Compare current elements
  - Copy smaller one down and increment that point

**Merge**

<table>
<thead>
<tr>
<th>L: 1 3 5 8</th>
<th>R: 2 4 6 7</th>
</tr>
</thead>
</table>

---

**Merge**

<table>
<thead>
<tr>
<th>↓</th>
<th>↓</th>
</tr>
</thead>
<tbody>
<tr>
<td>L: 1 3 5 8</td>
<td>R: 2 4 6 7</td>
</tr>
</tbody>
</table>

---

**Merge**

| 1 |

---

**Merge**

| 1 |

---
General algorithm:
- Keep a "pointer" (index) for where we are in each input array
- Start them both at the beginning
- Repeat until we're done:
  - Compare current elements
  - Copy smaller one down and increment that point
General algorithm:
- Keep a "pointer" (index) for where we are in each input array
- Start them both at the beginning
- Repeat until we're done:
  - Compare current elements
  - Copy smaller one down and increment that point
General algorithm:
- Keep a “pointer” (index) for where we are in each input array
- Start them both at the beginning
- Repeat until we’re done:
  - Compare current elements
  - Copy smaller one down and increment that point

What do we do now?
If we run off the end of either array, just copy the remaining from the other array
**MergeSort**

7 1 4 2 6 5 3 8

**MergeSort: implementation 1**

```java
mergeSort(data)
    if data.length <= 1
        return data
    else
        midpoint = data.length/2
        left = left half of data
        right = right half of data
        leftSorted = mergeSort(left)
        rightSorted = mergeSort(right)
        return merge(leftSorted, rightSorted)
```

**MergeSort: implementation 2**

```java
mergeSortHelper(data, low, high)
    if high-low > 1
        midPoint = low + (high-low)/2
        mergeSortHelper(data, low, mid)
        mergeSortHelper(data, mid, high)
    merge(data, low, mid, high)
```

What is the difference?
**Merge:**

merge(data, low, mid, high)

Assume:
- data starting at low up to, but not including, mid is sorted
- data starting at mid up to, but not including, high is sorted

Goal:
- data from low up to, but not including, high is sorted

**MergeSort**

```
mergeSort 7 1 4 2 6 5 3 8
mergeSort 7 1 4 2
mergeSort 6 5 3 8
mergeSort 7 1
mergeSort 4 2
mergeSort 6 5
mergeSort 3 8
mergeSort 1
mergeSort 4
mergeSort 2
mergeSort 6
mergeSort 5
mergeSort 3
mergeSort 8
```